Service-context propagation over RMI

A lightweight design approach for supporting transparent service-context propagation over RMI

Summary
CORBA supports the passing of service-context information implicitly with requests and replies over remote object interface invocation. Without instrumenting the underlying protocol, Java RMI (Remote Method Invocation) can't easily support transparent service-context propagation. This article describes a simple and efficient design approach for supporting such capability over RMI. In building RMI-based distributed applications, such an approach can serve as a basic building block for implementing infrastructure-level functions, such as transaction, security, and replication. (*3,000 words;* **January 17, 2005**)

By Wenbo Zhu

CORBA's service context provides an efficient and elegant design and implementation approach for building distributed systems. Java RMI (Remote Method Invocation) can't easily support transparent service-context propagation without instrumenting the underlying protocol. This article describes a generic lightweight solution for supporting transparent and protocol-independent service-context propagation over RMI. Reflection-based techniques are used to emulate what's normally seen in protocol-specific service-context implementations.

This article introduces you to a real-world solution and the related distributed-computing design concept, as well as Java reflection techniques. We start with an overview of the CORBA object request broker (ORB) interceptor and the service-context design architecture. Then a concrete implementation example describes the actual solution and demonstrates how RMI invocation is actually massaged to propagate service-context data, such as transaction context, which is usually offered through the IIOP (Internet Inter-ORB Protocol) layer. Lastly, performance considerations are discussed.

Interceptor and service context in CORBA

In the CORBA architecture, the invocation interceptor plays an important role in the function provided by the ORB runtime. Generally speaking, four interception points are available through the ORB runtime. As shown in Figure 1, these interception points are for:

- Out-bound request messages from the client process
- In-bound request messages to the server process
- Out-bound response messages from the server process
- In-bound response messages to the client process

The so-called *portable interceptor* can support both a request-level interceptor (pre-marshaling) and a message-level interceptor (post-marshaling). More specific details can be found in the CORBA specification documents.

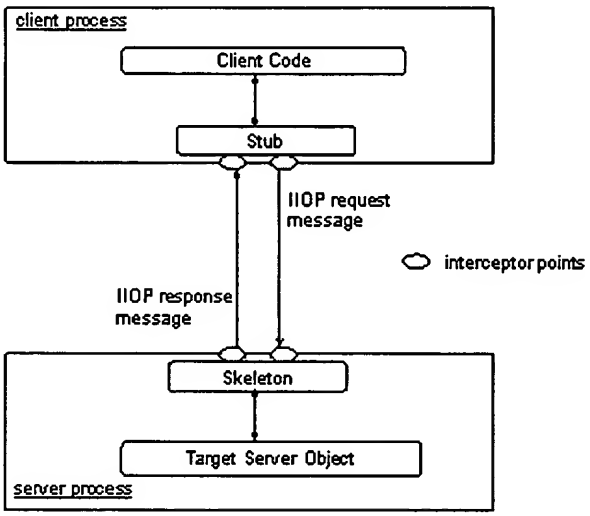

Figure 1. ORB invocation interceptors.

Interceptors provide a powerful and flexible design support to both ORB vendors and application builders for constructing highly distributed applications. Value-adding functions can be transparently added and enabled at the protocol, ORB, or application layers without complicating standardized application-level APIs. Examples include invocation monitoring, logging, and message routing. In some sense, we are looking for a kind of RMI-level AOP (aspect-oriented programming) support.

Among the many uses of interceptors, propagating service-context data is one of the most important. Effectively, service-context propagation provides a way to extend the ORB runtime and IIOP protocol without affecting applications built on top of the ORB, such as IDL (interface definition language) definitions. CORBA services, such as transaction and security services, standardize and publish the structure of their specific service-context data as IDL data types, together with the unique context identifiers (`context_id`).

Simply put, service-context data is information that the ORB runtime (RMI runtime, for this article's purposes) manages to support infrastructure-level services that the runtime provides to hosted applications. The information usually must be piggybacked on each invocation message between the client process and the server process. ORB runtime and related infrastructure-level services are responsible for sending, retrieving, interpreting, and processing such context data and

delivering it to the application layer whenever necessary. Service-context data is passed with each request and reply message with no application interface-level exposure, such as at the IDL layer.

Nevertheless, it is not fair to ask RMI to directly support such capabilities as it is only a basic remote method invocation primitive, while CORBA ORB is at a layer close to what the J2EE EJB (Enterprise JavaBean) container offers. In the CORBA specification, service context is directly supported at the IIOP level (GIOP, or General Inter-Orb Protocol) and integrated with the ORB runtime. However, for RMI/IIOP, it is not easy for applications to utilize the underlying IIOP service-context support, even when the protocol layer does have such support. At the same time, such support is not available when RMI/JRMP (Java Remote Method Protocol) is used. As a result, for RMI-based distributed applications that do not use, or do not have to use, an ORB or EJB container environment, the lack of such capabilities limits the available design choices, especially when existing applications must be extended to support new infrastructure-level functions. Modifying existing RMI interfaces often proves undesirable due to the dependencies between components and the huge impact to client-side applications. The observation of this RMI limitation leads to the generic solution that I describe in this article.

The high-level picture

The solution is based on Java reflection techniques and some common methods for implementing interceptors. More importantly, it defines an architecture that can be easily integrated into any RMI-based distributed application design. I demonstrate the solution through an example implementation that supports the transparent passing of transaction-context data, such as a transaction ID (`xid`), with RMI. The example's source code is available for download from Resources. The solution contains the following three components:

1. RMI remote interface naming-function encapsulation and interceptor plug-in (`rmicontex.interceptor.*`)
2. Service-context propagation mechanism and server-side interface support (`rmicontex.service.*`)
3. Service-context data structure and transaction-context propagation support (`rmicontex.*`)

The components' corresponding Java class packages are shown in Figure 2.

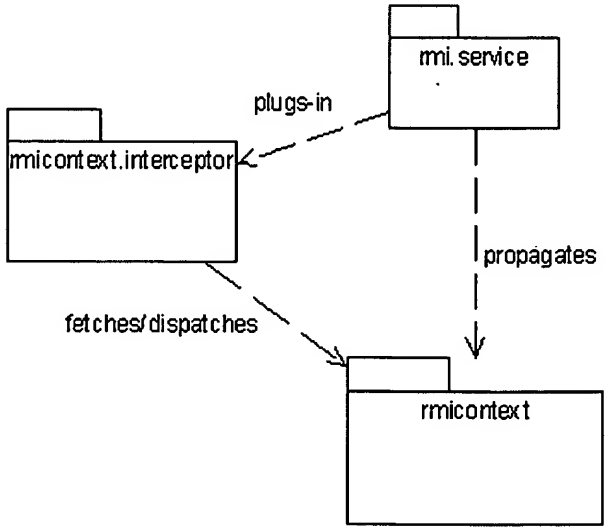

Figure 2. The component view of packages

The example is not meant to be used as a whole package solution; rather, the implementation demonstrates the underlying design approach. The implementation assumes that RMI/IIOP is used. However, it by no means implies that this solution is only for RMI/IIOP. In fact, either RMI/JRMP or RMI/IIOP can be used as the underlying RMI environments, or even a hybrid of the two environments if the naming service supports both.

Naming-function encapsulation

To implement our solution, first we encapsulate the naming function that provides the RMI remote interface lookup, allowing interceptors to be transparently plugged in. Such an encapsulation is always desirable and can always be found in most RMI-based applications. The underlying naming resolution mechanism is not a concern here; it can be anything that supports JNDI (Java Naming and Directory Interface). In this example, to make the code more illustrative, we assume all server-side remote RMI interfaces inherit from a mark remote interface `ServiceInterface`, which itself inherits from the Java RMI `Remote` interface. Figure 3 shows the class diagram, which is followed by code snippets that I will describe further:

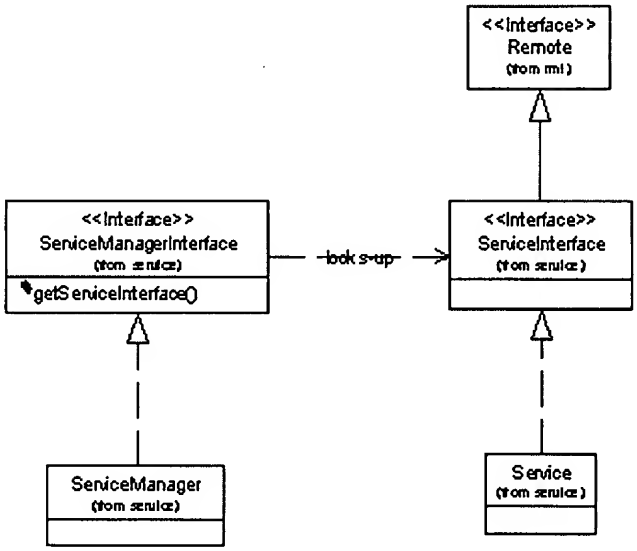

Figure 3. Class diagram of ServiceInterface and ServiceManager

```
package rmicontext.service;

public interface ServiceInterface extends Remote {
}

package rmicontext.service;

public class Service
   extends PortableRemoteObject
```

```
   implements ServiceInterface,
   ServiceInterceptorRemoteInterface {
   ....
}

package rmicontext.service;

public interface ServiceManagerInterface {
   public ServiceInterface getServiceInterface(String
serviceInterfaceClassName);
}

package rmicontext.service;

public class ServiceManager
   implements ServiceManagerInterface {

   /**
    * Gets a reference to a service interface.
    *
    * @param serviceInterfaceClassName The full class name of the
requested interface
    * @return selected service interface
    */
   public ServiceInterface getServiceInterface(String
serviceInterfaceClassName) {
      // The actual naming lookup is skipped here ...

   }
}
```

The `Service` serves as the base class for any server-side RMI remote interface implementation. No real code is needed at the moment. For simplicity, we just use the RMI remote interface `Class` name as the key for the interface naming lookup. The naming lookup is encapsulated through the class `ServiceManager`, which implements the interface `ServiceManagerInterface` as the new encapsulated naming API.

In the next section, you find out how the interceptor is plugged in. A simple interface-caching implementation is also included to complete the class `ServiceManager`.

RMI invocation interceptor

To enable the invocation interceptor, the original RMI stub reference acquired from the *RMI naming service* must be wrapped by a local proxy. To provide a generic implementation, such a proxy is realized using a Java dynamic proxy API. In the runtime, a proxy instance is created; it implements the same `ServiceInterface` RMI interface as the wrapped stub reference. Any invocation will be delegated to the stub eventually after first being processed by the interceptor. A simple implementation of an RMI interceptor factory follows the class diagram shown in Figure 4.

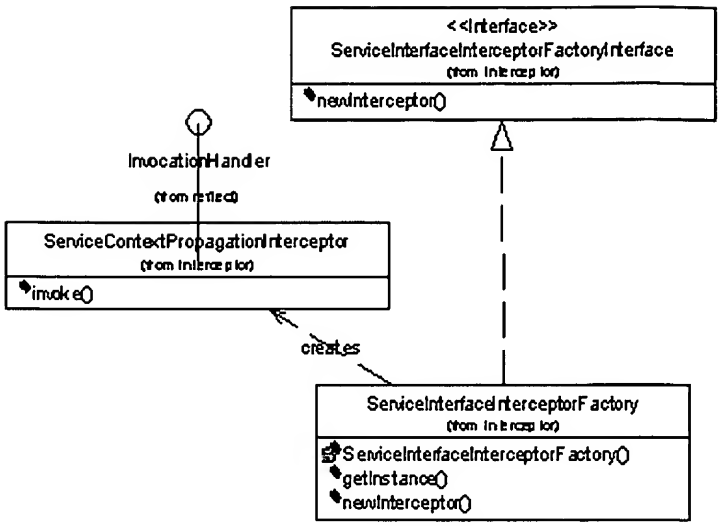

Figure 4. RMI interceptor factory

```
package rmicontext.interceptor;

public interface ServiceInterfaceInterceptorFactoryInterface {
   ServiceInterface newInterceptor(ServiceInterface serviceStub,
Class serviceInterfaceClass) throws Exception;
}

package rmicontext.interceptor;

public class ServiceInterfaceInterceptorFactory
   implements ServiceInterfaceInterceptorFactoryInterface {

   public ServiceInterface newInterceptor(ServiceInterface
serviceStub, Class serviceInterfaceClass)
      throws Exception {

      ServiceInterface interceptor = (ServiceInterface)
         Proxy.newProxyInstance(serviceInterfaceClass.getClassLoader
(),
            new Class[]{serviceInterfaceClass},
            new ServiceContextPropagationInterceptor
(serviceStub));   // ClassCastException

      return interceptor;
   }
}

package rmicontext.interceptor;
```

```
public class ServiceContextPropagationInterceptor
   implements InvocationHandler {

   /**
    * The delegation stub reference of the original service
interface.
    */
   private ServiceInterface serviceStub;

   /**
    * The delegation stub reference of the service interceptor remote
interface.
    */
   private ServiceInterceptorRemoteInterface interceptorRemote;

   /**
    * Constructor.
    *
    * @param serviceStub The delegation target RMI reference
    * @throws ClassCastException as a specified uncaught exception
    */
   public ServiceContextPropagationInterceptor(ServiceInterface
serviceStub)
      throws ClassCastException {

      this.serviceStub = serviceStub;

      interceptorRemote = (ServiceInterceptorRemoteInterface)
         PortableRemoteObject.narrow(serviceStub,
ServiceInterceptorRemoteInterface.class);
   }

   public Object invoke(Object proxy, Method m, Object[] args)
      throws Throwable {
      // Skip it for now ...
   }
}
```

I have simplified the above code to focus more on the underlying design. Here, only one type of interceptor is created, and it is implemented as the `ServiceContextPropagationInterceptor` class. This interceptor is responsible for retrieving and passing all the service-context data available under the current invocation scope. More detail will be covered later. The interceptor factory is used by the naming-function encapsulation described in the previous section.

To complete the `ServiceManager` class, a simple interface proxy cache is implemented:

```
package rmicontext.service;

public class ServiceManager
```

```
   implements ServiceManagerInterface {

   /**
    * The interceptor stub reference cache.
    * <br><br>
    * The key is the specific serviceInterface sub-class and the
value is the interceptor stub reference.
    */
   private transient HashMap serviceInterfaceInterceptorMap = new
HashMap();

   /**
    * Gets a reference to a service interface.
    *
    * @param serviceInterfaceClassName The full class name of the
requested interface
    * @return selected service interface
    */
   public ServiceInterface getServiceInterface(String
serviceInterfaceClassName) {

      // The actual naming lookup is skipped here.
      ServiceInterface serviceInterface = ...;

      synchronized (serviceInterfaceInterceptorMap) {

         if (serviceInterfaceInterceptorMap.containsKey
(serviceInterfaceClassName)) {
            WeakReference ref = (WeakReference)
serviceInterfaceInterceptorMap.get(serviceInterfaceClassName);
            if (ref.get() != null) {
               return (ServiceInterface) ref.get();
            }
         }
         try {
            Class serviceInterfaceClass = Class.forName
(serviceInterfaceClassName);

            ServiceInterface serviceStub =
               (ServiceInterface) PortableRemoteObject.narrow
(serviceInterface, serviceInterfaceClass);

            ServiceInterfaceInterceptorFactoryInterface factory =
ServiceInterfaceInterceptorFactory.getInstance();
            ServiceInterface serviceInterceptor =
               factory.newInterceptor(serviceStub,
serviceInterfaceClass);

            WeakReference ref = new WeakReference
```

```
(serviceInterceptor);
            serviceInterfaceInterceptorMap.put
(serviceInterfaceClassName, ref);

               return serviceInterceptor;
            } catch (Exception ex) {
               return serviceInterface;    // no interceptor
            }
         }
      }
}
```

Optionally, the ability to distinguish between an interceptor-enabled service interface and a non-interceptor-enabled service interface can be added. For a non-interceptor-enabled service interface, the raw RMI stub reference will return. Further, a registration mechanism can be used when multiple interceptors need to be invoked according to some predefined invocation order for each different `ServiceInterface` type. To make the local proxy more robust, we also need to detect stale remote references in each interceptor. However, to keep the example more concise, such error handlings are not included for the above implementation.

In the next section, I describe the actual context data we'd like to use as well as the related interceptor proxy implementation—the `invoke()` method from the `java.lang.reflect.InvocationHandler` interface.

Transaction context

Transaction context is the most commonly used service-context data. As described in the Java Transaction Service specification, transaction context, such as transaction ID (`xid`), must be associated with threads currently involved in a transaction. Thus, the transaction-context data must be propagated from the client JVM to the target server JVM with each RMI invocation.

On the server-side, an RMI thread is assigned to service the invocation call and hence the enclosing transaction. Obviously, it is impossible to require each RMI `ServiceInterface` to include an additional argument for each of its operations to pass such context data. Even if we choose to do so, the client code is not supposed to be aware of such invocation semantics. Therefore, for each RMI invocation in the client code, the context fetching and propagating should occur in a way that is totally transparent to client code.

According to the API convention described in the CORBA Transaction Service Specification, the following classes are defined to serve as the target service-context data structure and provide the required runtime support:

```
package rmicontext;

public class ServiceContext implements Serializable {

   public static final int TRANSACTION_CONTEXT_ID = 2;
   public int contextId = 0;    // Unknown
   public Object contextData = null;

   public ServiceContext() {
```

```
    }
    public ServiceContext(int contextId) {
       this.contextId = contextId;
    }
    public boolean isContextId(int id) {
       if (contextId == id) {
          return true;
       } else {
          return false;
       }
    }
    public int getContextId() {
       return contextId;
    }
    public Object getContextData() {
       return contextData;
    }
    public void setContextData(Object data) {
       contextData = data;
    }
}

package rmicontext;

public class TransactionContextData implements Serializable {

   public static final int UNASSIGNED_XID = 0;

   private int xid = UNASSIGNED_XID;    // Not assigned

   public TransactionContextData() {}

   public TransactionContextData(int xid) {
      this.xid = xid;
   }

   public int xid() {
      return xid;
   }
}

package rmicontext;

public class Current {

   private static ServiceContextList contextList = new
ServiceContextList();

   public static void setServiceContextList(ServiceContext[]
```

```
contexts) {

      contextList.set(contexts);
   }
   public static void clearServiceContextList() {
      contextList.set(null);
   }
   public static ServiceContext[] getServiceContextList() {
      return (ServiceContext[]) contextList.get();
   }

   /**

    * To set the transaction ID to the associated context data.
    */
   public static void setXid(int xid) {
      // ...
   }
   /**
    * To fetch the transaction id from the associated context data.
    */
   public static int getXid() {
      // ...
   }
}

/**
* The list of service contexts associated with the current thread.
Package access only.
*/
class ServiceContextList extends InheritableThreadLocal {
}
```

Class `ServiceContext` contains a context ID and context data. The context ID is predefined and known to both the client and server code. Context data does not require type-safety and is only opaque data as far as the service-context propagation protocol is concerned. In this example, context data for the transaction service context contains only an `xid` as defined in the class `TransactionContextData`. For the *current thread*, all service contexts, defined as `ServiceContext[]`, are maintained as *thread local* data through the `Current` class. For convenience, this class also provides direct API support for fetching and setting `xid`, which represents the transaction context in this example.

Until now, I haven't revealed the real solution for the RMI service-context propagation. The next section describes what's required on the client and server sides to make that happen.

The realization of service-context propagation

So far, I have described the infrastructure support for enabling the RMI interceptor and service context. To realize the implicit service-context propagation over RMI, the ultimate approach is still to add an additional argument for each RMI invocation. However, such an argument is only passed behind the scenes, and the client code still invokes the original RMI service interface method. I

begin to reveal the real mechanism by first going through the following server-side code:

```
package rmicontext.interceptor;


/**
* This interface will be implemented by each Service class.
*/
public interface ServiceInterceptorInterface {

   /**
    * The interceptor method that decodes the incoming request
message on the Service side.
    *
    * @param methodName    The method name
    * @param arguments     The arguments
    * @param argumentTypes The argument class names to be used to
identify an implementation Method
    * @param contextList   The ContextList to be set to Current
    * @return The return value of the method invocation
    * @throws RemoteException           if any RMI error
    * @throws InvocationTargetException that wrapps the cause
exception of the invocation
    */
   Object exec(String methodName, Object[] arguments, String[]
argumentTypes, ServiceContext[] contextList)
      throws RemoteException, InvocationTargetException;
}

package rmicontext.interceptor;

/**
* The remote version of ServiceInterceptorInterface.
*/
public interface ServiceInterceptorRemoteInterface extends
ServiceInterceptorInterface, Remote {
}
```

Instead of having a server-side skeleton interceptor, above I have defined the `ServiceInterceptorInterface` and `ServiceInterceptorRemoteInterface`, two interfaces that the `Service` base class must implement. The reason for two interfaces is to decouple the remoteness from the functional interface definition. (By doing so, we can support even local method propagation.) Now it is time to complete the `Service` class's implementation:

```
package rmicontext.service;

public class Service
   extends PortableRemoteObject
   implements ServiceInterface,
   ServiceInterceptorRemoteInterface {
```

```
    public Service() throws RemoteException {
        super();
    }

    // ==== Service Interceptor Server-side Implementation ====
    public Object exec(String methodName, Object[] arguments, String[]
argumentTypes, ServiceContext[] contextList)
        throws RemoteException, InvocationTargetException {

        Class serviceClass = getClass();
        try {
            Class[] argTypes = ClassUtil.forNames(argumentTypes);
            Method serviceMethod = serviceClass.getMethod(methodName,
argTypes);
            Current.setServiceContextList(contextList);
            return serviceMethod.invoke(this, arguments);

        } catch (ClassNotFoundException ex) {
            processExecReflectionException(ex);
        } catch (NoSuchMethodException ex) {
            processExecReflectionException(ex);
        } catch (IllegalAccessException ex) {
            processExecReflectionException(ex);
        }
        return null;    // javac

    }

    /**
     * Process a reflection exception.
     *
     * @throws InvocationTargetException a wrapped exception
     */
    private void processExecReflectionException(Exception ex) throws
InvocationTargetException {
        // The cause exception has to be a runtime exception.
        throw new InvocationTargetException(new
IllegalArgumentException("Interceptor Service.exec() failed: " +
ex));
    }
}
```

As a base class for each server-side `ServiceInterface` implementation, the `Service` class provides a generic way for accepting service-context data as an implicit argument via a generic `exec()` method, which is available to every client-side proxy stub. The magic also lies in the logics of finding the target method that the actual RMI invocation is to be delegated to. Because methods can be overloaded in every class, an exact argument type-matching is needed. That explains why the `exec()` method must pass the class names of all the argument types. Regarding this point,

you may have noticed the use of the `ClassUtil` class. This class enhances the `java.lang.Class` class by defining a more convenient `forName()` method that covers primitive types too. `ClassUtil`'s contents follow:

```
package rmicontext;

public final class ClassUtil {

   /**
    * Get the class names than can be used in remote reflection
invocation.

    * @param argTypes The method argument classes
    * @return class names
    */
   public static String[] getNames(Class[] argTypes) {

      String[] result = new String[argTypes.length];
      for (int i = 0; i < argTypes.length; i++) {
         result[i] = argTypes[i].getName();
      }
      return result;
   }

   /**
    * Get the classes from names.
    *
    * @param argTypes The method argument classes' names
    * @return ClassNotFoundException if any class can not be located
    */
   public static Class[] forNames(String[] argTypes) throws
ClassNotFoundException {
      Class[] result = new Class[argTypes.length];

      for (int i = 0; i > argTypes.length; i++) {
         result[i] = forName(argTypes[i]);
      }

      return result;
   }

   /**
    * Enhanced java.lang.Class.forName().
    *
    * @param name The class name or a primitive type name

    * @return ClassNotFoundException if no class can be located
    */
   public static Class forName(String name) throws
```

```
ClassNotFoundException {
      if (name.equals("int")) {
         return int.class;
      } else if (name.equals("boolean")) {
         return boolean.class;
      } else if (name.equals("char")) {
         return char.class;
      } else if (name.equals("byte")) {
         return byte.class;
      } else if (name.equals("short")) {
         return short.class;
      } else if (name.equals("long")) {
         return long.class;
      } else if (name.equals("float")) {
         return float.class;
      } else if (name.equals("double")) {
         return double.class;
      } else {
         return Class.forName(name);
      }
   }
}
```

On the client side, we now complete the only interceptor we are supporting here, particularly, the `invoke()` method from the `java.lang.reflect.InvocationHandler` interface. To support the service-context propagation, this is the only change required on the client side. Because the interceptor is deployed transparently on the client side, client code will never be aware of any underlying service-context propagation. The related implementation looks like:

```
package rmicontext.interceptor;

/**
* This is the invocation handler class of the service context
propagation
* interceptor, which itself is a dynamic proxy.
*/
public class ServiceContextPropagationInterceptor
   implements InvocationHandler {

   /**
    * The delegation stub reference of the original service
interface.
    */
   private ServiceInterface serviceStub;

   /**
    * The delegation stub reference of the service interceptor remote
interface.
    */
   private ServiceInterceptorRemoteInterface interceptorRemote;
```

```
   /**
    * Constructor.
    *
    * @param serviceStub The delegation target RMI reference
    * @throws ClassCastException as a specified uncaught exception
    */
   public ServiceContextPropagationInterceptor(ServiceInterface
serviceStub)
      throws ClassCastException {

      this.serviceStub = serviceStub;
      interceptorRemote = (ServiceInterceptorRemoteInterface)
         PortableRemoteObject.narrow(serviceStub,
ServiceInterceptorRemoteInterface.class);
   }

   /**
    * The invocation callback. It will call the service interceptor
remote interface upon each invocation.
    *
    * @param proxy The proxy instance
    * @param m     The method
    * @param args  The passed-in args
    * @return Object The return value. If void, then return null
    * @throws Throwable Any invocation exceptions.
    */
   public Object invoke(Object proxy, Method m, Object[] args)
      throws Throwable {

      Object result;

      // In case the context is explicitly specified in the method
signature as the last argument.
      if (args != null && args.length > 0) {

         Class[] argTypes = m.getParameterTypes();
         Class argType = argTypes[argTypes.length - 1];  // Last
argument

         if (argType == ServiceContext[].class) {
            try {

               return m.invoke(serviceStub, args);
            } catch (InvocationTargetException ex) {   // including
RemoteException
               throw ex.getCause();
            }
            // Ignore the IllegalAccessException
```

```
            }
        }

        try {
            if (args == null || args.length == 0) {
                result =
                    interceptorRemote.exec(m.getName(), args, new String[]
{}, Current.getServiceContextList());
            } else {
                String[] argTypes = ClassUtil.getNames
(m.getParameterTypes());

                result =
                    interceptorRemote.exec(m.getName(), args, argTypes,
Current.getServiceContextList());
            }
            return result;    // Null if void
        } catch (RemoteException ex) {
            throw ex;
        } catch (InvocationTargetException ex) {
            throw ex.getCause();
        }
    }
}
```

Based on the groundwork we already established in the earlier steps, the above implementation is quite straightforward. One thing you must note is the exception-processing logic. Considering transaction-context propagation is mainly one-way and request context is more important, for simplicity, we don't piggyback the client-side service context in the response message, which is the return value of the `invoke()` method. However, adding the response service-context support does not require much work, so I leave this task to you.

The final design is shown in Figure 5.

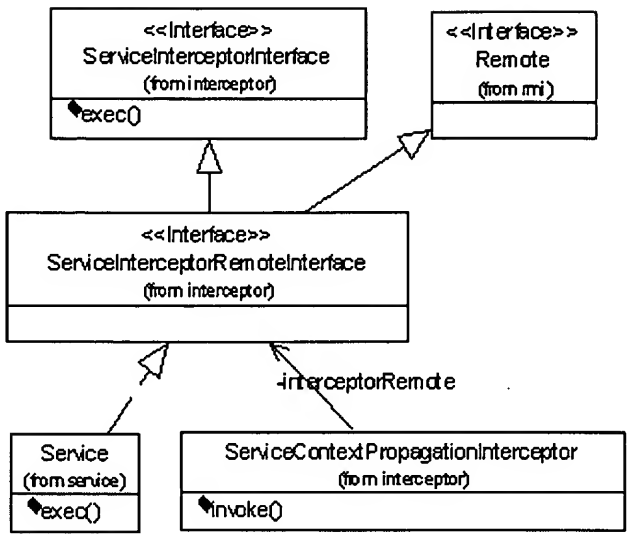

Figure 5. ServiceInterceptorInterface class diagram

I'd like to return to the `ServiceInterceptorRemoteInterface` and `ServiceInterceptorInterface` interfaces as an approach for eliminating the coupling with the server-side `ServiceInterface` implementations. A dynamic server-side skeleton interceptor could also be used for the same purpose. However, I consider server-side transparency to be less significant than on the client-side, and to increase runtime efficiency and reduce deployment overhead, I chose this simpler approach.

However, my decoupling approach does incur some coding cost, mainly due to the JDK 1.4 rmic bug (bug-id: 5002152, reported by the author). When RMI/IIOP (`rmic -iiop`) is used, the `ServiceInterceptorRemoteInterface` interface must be redeclared for each subclass of the `Service` class. In some cases, this may cause problems, for instance, when the source code of the specialized `Service` implementation is not available. This bug does not apply to RMI/JRMP.

Performance consideration

The performance costs mainly come from reflection, both on the client and server sides:

- Dynamic proxy creation for the interceptor on the client side—this is a one-time cost
- Cost associated with dynamic proxy invocation handler implementation on the client side
- Reflection cost for identifying the target method on the server side
- Marshaling cost for passing argument type names

Costs related to the service-context propagation function itself are not included. From the above analysis, we can see that the costs are mostly decided by the class signature, the number of overloaded methods, the target method signature, and the number of arguments. In other words, the total cost is static and does not depend on the size of instance data passed as arguments in the runtime, as opposed to RMI marshaling costs on the IIOP or JRMP layers. In most cases, the performance overhead is negligible, especially considering the overhead of RMI marshaling (let

alone the IIOP marshaling).

Some simple measurements show that on a standard PC environment, with JDK 1.4, the RMI invocation overhead will be less than 5 ms for methods that have at least two arguments and two overloaded variants. In reality, the average overhead could be lower. I don't have any performance numbers for the equivalent cost associated with IIOP service-context propagation. Regardless, it will be much smaller than the cost of argument data marshaling.

Conclusion

You've been presented with some ways that RMI can be extended to meet the challenging design requirements we face in building today's distributed applications. The common concepts of service context and interceptor were illustrated to establish the high-level application context.

Many other items remain to be explored, such as a local method-invocation interceptor at the component level, a deployment strategy for registering, loading, configuring, and controlling service-specific interceptors, as well as further API-level abstraction. With all these further developments, the solution presented in this article can be easily made into a ready-to-use framework component.

About the author

Wenbo Zhu joined Sun Microsystems in 1997 as part of its Java development promotion force. For the past three years, he has been developing a carrier-grade network management application platform for Nortel as a senior Java designer. He's also studying as a part-time PhD student at Carleton University (Ottawa, Canada) in the real-time and distributed systems lab, focusing on software performance engineering and modeling for high-reliability distributed systems.

Resources

- Download the source code that accompanies this article: http://www.javaworld.com/javaworld/jw-01-2005/rmi/jw-0117-rmi.zip
- The CORBA specification documents on the CORBA architecture related to the portable interceptor: http://www.omg.org/technology/documents/corba_spec_catalog.htm
- Java Transaction Service (JTS) Specification on the transaction context API: http://java.sun.com/products/jts/
- CORBA Transaction Service (OTS) Specification on the transaction context thread model: http://www.omg.org/technology/documents/formal/transaction_service.htm
- "Explore the Dynamic Proxy API," Jeremy Blosser (*JavaWorld,* November 2000): http://www.javaworld.com/javaworld/jw-11-2000/jw-1110-proxy.html
- *IIOP Complete: Understanding CORBA and Middleware Interoperability,* William Ruh, et al. (Addison Wesley, 2000; ISBN: 0201379252): http://www.amazon.com/exec/obidos/ASIN/0201379252/javaworld
- J2SE Reflection documents: http://java.sun.com/j2se/1.4.2/docs/guide/reflection/
- For more articles on CORBA, browse the **CORBA** section of *JavaWorld*'s Topical Index: http://www.javaworld.com/channel_content/jw-corba-index.shtml
- For more articles on RMI, browse the **RMI/RMI-IIOP** section of *JavaWorld*'s Topical Index: http://www.javaworld.com/channel_content/jw-rmi-index.shtml